URL: http://www.elsevier.nl/locate/entcs/volume70.html 17 pages

Junctive Compositions of Specifications in Total and General Correctness

Steve Dunne

School of Computing and Mathematics, University of Teesside
Middlesbrough, TS1 3BA, UK
email: s.e.dunne@tees.ac.uk

Abstract

First against a conventional total-correctness background of wp semantics, we define a small family of compositions for predicate-pair specifications based on simple logical conjunction and disjunction, discussing the formal properties and offering an intuitive operational interpretation of each. Three of these compositions we recognise as familiar; the fourth one, our *concert*, is new but lacks any apparent use. Then we re-interpret our compositions in the context of general correctness to very useful effect.

1 Introduction

There has been significant interest over the last decade in augmenting the Refinement Calculus of Back [2], Morgan [16] and Morris [17] with a suitable range of specification composition operators to support the incremental construction of large specifications from component specifications. A frequently cited contribution is that of Ward [19] which was inspired by Z's schema calculus operators [18,20]. More recently, Mahony [14] has formulated a more generic approach to lifting the relational operators of a model-based specification formalism like Z into the Refinement Calculus's semantic realm of predicate transformers, in a way which preserves the notion of promotion so useful in Z for building specifications incrementally. In the meantime others have sought to analyse and classify such specification compositions within an appropriate theoretical framework. In particular, the work of Back and Butler [3] has been influential in providing such a framework, while Leino and Manohar [13] have undertaken a close analysis of join compositions.

In this paper we address the way in which the underlying correctness semantics we adopt for our specifications influences how we can formulate such compositions, and subsequently how we can interpret and use them. In particular, we examine a small simply definable family of so-called *junctive* compositions. We immediately recognise three of these as special cases of ones already

defined by others. The remaining one, which we call *concert*, is new but of no apparent use. Surprisingly, though, when we shift our correctness perspective from total correctness to general correctness not only do the definitions of the individual compositions of our family become more uniformly simple, but our hitherto useless concert acquires genuine utility.

In Section 2 we review the orthodox total-correctness basis which Dijkstra's wp semantics provides for the predicate-pair specifications of the Refinement Calculus in its various manifestations. In Section 3 we formulate our family of junctive compositions, discussing their properties and relating them to those of others. In Section 4 we describe general correctness, explaining the alternative semantic basis it provides for interpreting our predicate-pair specifications and programs. In Section 5 we re-interpret our family of junctive compositions in this new semantic context. Finally, in Section 6 we discuss in particular how our concert composition can now be usefully applied.

2 Specifications in Total Correctness

The use of a pair of predicates in representing a software specification is ubiquitous in the refinement literature. We have in [15], for example, the precondition P and postcondition Q of the Morgan specification statement

$$w : [P , Q]$$

acting on a frame w of variables, and in [10] the assumption P and commitment Q of the Hoare and He design

$$P \vdash Q$$

while in [3] we have the assertion predicate P and demonic update relation Q of Back and Butler's normal form

$$\{P\} ; [Q]$$

of a conjunctive predicate transformer. In B [1,6] we can extract the precondition P and before-after predicate Q of a generalised substitution acting on a frame w of variables, allowing us then to express that substitution in Abrial's normal form

$$P \mid \mathbb{Q}\, w'.\ Q \implies w := w'$$

where w' is the frame obtained by systematically dash-decorating all the variables of w.

In all these representations P is a simple "unary" predicate, or *condition*, over the state space which serves to characterise the realm of applicability of the specification, while Q is generally a "binary" predicate relating before-states to after-states and thus characterising which particular after-states are

allowed by the specification from which particular applicable before-states[1]. Such a predicate pair can be informally interpreted as describing an abstract program such that

> If the program starts in circumstances satisfying P, it will terminate in a state satisfying Q (having modified only variables of w).

Non-applicable before-states may also be admitted by the predicate Q but are irrelevant to the meaning of the specification. In fact the specification places no constraint on what should happen if the program is started in a state not satisfying P. The program may behave quite unpredictably in such circumstances: it might not terminate; equally, it might terminate anarchically in an after-state unconstrained by Q.

2.1 Infeasibility

On the other hand, if not all the applicable states defined by P are admitted by Q as before-states the specification is partial. A partial specification lacks enough information to permit it to be individually implemented, and as such is said to be *infeasible*. Nevertheless, two or more infeasible specifications can be combined through the important composition mechanism of demonic choice to make a feasible one.

2.2 Weakest feasible completion

We can modify an infeasible specification into a feasible one by sufficiently strengthening its applicability predicate. We call the resulting specification the *weakest feasible completion* of the original one. For example, the weakest feasible completion of the Hoare-He design $P \vdash Q$ in the context of a global frame v of variables is $P \wedge (\exists\, v' \bullet Q) \vdash Q$.

2.3 Weakest Precondition Semantics

So much for our informal interpretations of predicate-pair specifications. Their meanings are more formally expressed by Dijkstra's weakest-precondition (wp) predicate-transformer semantics [5]. This is a semantics of total correctness, so called because it is concerned only with what can be unconditionally guaranteed about the outcome of a computation. For example, the weakest precondition for the Hoare-He design $P \vdash Q$ to establish any given postcondition R is given by

$$wp(P \vdash Q\,, R) \quad =_{df} \quad P \;\wedge\; (\forall\, v' \bullet Q \Rightarrow R') \qquad 2.3.1$$

[1] Symbolic conventions for Q differ from one formalism to another: in a Morgan specification statement, for example, the before-state is expressed by 0-subscripted variables, while in a He-Hoare design or a generalised substitution the after-state is expressed by dashed variables.

where v is the global frame of all variables in scope, and R' signifies that variant of R in which all occurrences of the variables of v have been replaced by corresponding ones of v'.

Equally, when we interpret a Back and Butler normal form $\{P\}\,;\,[Q]$ as a specification we do so by interpreting the conjunctive predicate transformer it denotes as a wp predicate transformer. In the wp calculus an infeasible specification manifests as sometimes able to establish any postcondition at all, even false. We therefore say an infeasible specification exhibits miraculous behaviour.

2.4 Problems in Defining Specification Compositions

Unfortunately, it is inherent in our total-correctness semantics that predicate pairs do not provide a unique canonical form for specifications. As we have already noted, there is potential redundancy in the second predicate since its before-values may extend outside the first predicate, such inapplicable before-values being irrelevant to the meaning of the specification. Thus for example, the Hoare-He design $P \vdash Q$ is semantically equivalent both to $P \;\vdash\; P \Rightarrow Q$ and to $P \;\vdash\; P \wedge Q$. Indeed, more generally it is equivalent to $P \vdash R$, where R is *any* predicate such that

$$P \wedge Q \;\;\leq\;\; R \;\;\leq\;\; P \Rightarrow Q$$

in the predicate implication ordering $\leq$ whose bottom is false and top true.

This can cause problems when defining a new specification composition if we base our definition on such representations of our specifications. Consider the following "definition" of a join-type operator "$*$" on Hoare-He designs:

$$P_1 \vdash Q_1 \;*\; P_2 \vdash Q_2 \quad =_{df} \quad P_1 \vee P_2 \;\vdash\; Q_1 \wedge Q_2$$

In fact, our $*$ operator is not well-defined since it doesn't uphold Leibniz's equality rule that if $S1 = S2$ then $S1 * S = S2 * S$. For example, $\mathit{false} \vdash \mathit{true}$ and $\mathit{false} \vdash \mathit{false}$ are alternative representations the same abortive design, yet

$$\begin{aligned} \mathit{false} \vdash \mathit{true} \;*\; \mathit{true} \vdash \mathit{true} \quad &= \quad \mathit{false} \vee \mathit{true} \;\vdash\; \mathit{true} \wedge \mathit{true} \\ &= \quad \mathit{true} \;\vdash\; \mathit{true} \end{aligned}$$

which is clearly feasible, while

$$\begin{aligned} \mathit{false} \vdash \mathit{false} \;*\; \mathit{true} \vdash \mathit{true} \quad &= \quad \mathit{false} \vee \mathit{true} \;\vdash\; \mathit{false} \wedge \mathit{true} \\ &= \quad \mathit{true} \;\vdash\; \mathit{false} \end{aligned}$$

which we recognise as the everywhere infeasible specification *magic*.

Such definitional mistakes are insidious. For example, in [3] Back and Butler briefly consider an alternative fusion operator $\oplus$ for conjunctive commands, "defined" by

$$\{p_1\}\,;\,[Q_1] \;\oplus\; \{p_2\}\,;\,[Q_2] \quad =_{df} \quad \{p_1 \vee p_2\}\,;\,[Q_1 \wedge Q_2]$$

They observe that

> This operator results in a true co-refinement .... However it is not the least conjunctive co-refinement so that use of this operator could result in specifications being strengthened unnecessarily when being combined. More seriously, this operator is not guaranteed to preserve refinement of its operands.

They go on to offer an example which illustrates the latter defect, and finally conclude that "these weaknesses rule $\oplus$ out as a useful co-refinement operator". Of course, that they even proceeded this far in their consideration of it betrays that they overlooked their $\oplus$ is ill-defined to begin with, for the same reason as our above $*$ operator for designs.

The problem can be countered by first *normalising* our predicate pairs so that the second predicate is weakened as far as is consistent with preserving the total-correctness meaning of the specification. For example, the normalised form of the Morgan specification statement $w : [P, Q]$ is $w : [P, P_0 \Rightarrow Q]$ where P_0 signifies that variant of P in which all occurrences of the variables of w have been replaced by corresponding ones of w_0, while that of the Hoare-He design $P \vdash Q$ is $P \vdash P \Rightarrow Q$.

2.5 Refinement in TotalCorrectness

A total-correctness specification T of course refines another one S, signified by writing $S \sqsubseteq T$, if T can always guarantee establish any given postcondition R in circumstances where S can. Formally

$$S \sqsubseteq T \quad =_{df} \quad wp(S, R) \leq wp(T, R) \quad \text{for every postcondition } R$$

Given definition 2.3.1 this means that a total-correctness predicate-pair specification such as the design $P_1 \vdash Q_1$ is refined by another one $P_2 \vdash Q_2$ precisely if $P_1 \leq P_2$ and $P_1 \wedge Q_2 \leq Q_1$.

3 Junctive Compositions in Total Correctness

By the term *junctive* composition we mean a composition of predicate-pair specifications obtained by simultaneously conjoining or disjoining their respective first and second predicates. Clearly this gives us four such compositions in all. In the case of disjoined first predicates we take the precaution of normalising second predicates into their weakest forms before conjoining or disjoining them, so as to avoid the kind of well-definedness problem we exposed in Section 2.4.

From now on we will work in the notation of Hoare-He designs, assumed to share a common alphabet of variables. We will also restrict our attention to binary compositions, though each of them could be generalised in an obvious way to apply to arbitrary sets of, rather than just pairs of, designs. The four resulting compositions of the designs $P_1 \vdash Q_1$ and $P_2 \vdash Q_2$ appear in Table 1.

symbol	name	definition
$[]$	demonic choice	$P_1 \wedge P_2 \ \vdash \ Q_1 \vee Q_2$
$\odot$	fusion	$P_1 \wedge P_2 \ \vdash \ Q_1 \wedge Q_2$
$\diamond$	join	$P_1 \vee P_2 \ \vdash \ (P_1 \Rightarrow Q_1) \wedge (P_2 \Rightarrow Q_2)$
$\#$	concert	$P_1 \vee P_2 \ \vdash \ (P_1 \Rightarrow Q_1) \vee (P_2 \Rightarrow Q_2)$

Table 1
Total-correctness junctive compositions of $P_1 \vdash Q_1$ with $P_2 \vdash Q_2$

It is important to note that we could have also defined the first two compositions in Table 1 using the same longer normalised forms $P_1 \Rightarrow Q_1$ and $P_2 \Rightarrow Q_2$ of their operands' commitments that feature in the definitions of the last two compositions in the table. This would have made no difference to the meanings of the first two compositions. The use of these longer forms in defining the last two compositions of Table 1 is, on the other hand, crucial for well-definedness.

Besides being well-defined, the four compositions of Table 1 are also all monotonic with respect to total-correctness refinement. Back and Butler [3] stress, as have others, that this is an essential property since it allows for piecewise refinement of programs. Mahony [14] refers to the rather stronger property of *compositionality* which requires in addition that the operator is part of the implementation code language, so that using it to combine fragments of code yields code.

3.1 Demonic Choice

The demonically nondeterministic choice operator $[]$ of Table 1 is very familiar. Our operational intuition of $[]$ when applied to feasible operands is that it represents a choice of executions entirely outside our control, being vested as we imagine in a demon who inhabits the machine on which our program is being executed. Nevertheless, the demon is constrained by feasibility in the sense that when faced with a choice between feasible and infeasible alternatives, he is obliged to choose the feasible one.

3.2 Fusion

The fusion operator $\odot$ of Table 1 is actually a specialisation to specifications (i.e. conjunctive predicate transformers) of that defined more generally by Back and Butler [3] for any pair of predicate transformers with the same signature. Our intuition of $\odot$ is that it constrains the nondeterminism of the individual terminating behaviours of each of its operands where these are jointly guaranteed to terminate, so as to achieve an agreed outcome. Where this is impossible because intersection of those behaviours is empty the fusion is infeasible and so there behaves miraculously. Where either operand isn't guaranteed to terminate, neither is their fusion.

3.3 Join

The join operator $\diamond$ of Table 1 is the binary version of that defined by Leino and Manohar [13]. It is in fact the least co-refinement of the two specifications concerned: that is, the weakest specification which refines both of them. Leino and Manohar argue that such a composition is important because it provides a mechanism for reconciling inadvertently disparate specifications of the same component which have already assumed by different potential users of the component, and as the basis for a paradigm of software construction by parts.

3.4 Concert

The concert operator # of Table 1 is new here in our current context of total correctness. We can imagine it being implemented, for example, by the unix fork mechanism. Our operational intuition of # is that it represents parallel execution of its two operands on disjoint copies of the global state in a termination pact: these two executions proceed until either terminates, upon which the resulting actual global state is recovered entirely from the terminating one's copy. The other execution is aborted, its copy of the global state being discarded. It is difficult to conceive any practical application of such a composition. Although it incurs a significantly greater operational overhead, it is scarcely any more predictable than $[]$. Where neither or both its operands are guaranteed to terminate # behaves exactly as $[]$. Where only one is guaranteed to terminate #'s termination though guaranteed will have an unpredictable accompanying result, since the predictably terminating operand's termination may be pre-empted by the anarchic termination of the other operand.

3.5 Ward's Specification Constructors

Ward [19] was inspired by Z's schema disjunction and conjunction operators to formulate, in terms of wp, new analogous compositions for total-correctness specifications in general. These have similarities with ours. Indeed, his subsidiary *miraculous conjunction* operator is identical to our $\odot$ in Table 1, but

in Ward's case this is merely a stepping stone to his final *conjunction* $S \sqcap T$ of total-correctness specifications S and T, obtained by taking the weakest feasible completion of $S \odot T$. Since taking weakest feasible completions is non-monotonic, Ward's conjunction doesn't preserve refinements.

His other main operator, the *disjunction* $S \sqcup T$ of S and T is characterised, to use the idiom of Dijkstra's Guarded Command Language [5], by

$$S \sqcup T \quad = \quad \textbf{if } Pre_S \; - \!\!\rightarrow S \; [] \; Pre_T \; - \!\!\rightarrow T \; \textbf{ fi}$$

where Pre_S and Pre_T denote the respective applicability predicates of S and T. Thus Ward's disjunction seems to be a sanitized version of our # with no risk of anarchic termination as long as we execute it within the applicability of either of its operands. In the notation of Hoare-He designs we can express it by

$$P_1 \vdash Q_1 \quad \sqcup \quad P_2 \vdash Q_2 \qquad =_{df} \qquad P_1 \vee P_2 \; \vdash \; (P_1 \wedge Q_1) \vee (P_2 \wedge Q_2)$$

Unfortunately, as Ward himself concedes, his disjunction $\sqcup$ shares with his conjunction $\sqcap$ the same defect of not being monotonic with respect to refinement. Tantalisingly then, the formulation of a genuinely useful concert-like operator seems to remain just beyond our grasp. In its original form our # is too susceptible to anarchic termination to be useful, but if we attempt to tame it as in Ward's $\sqcup$, we seem condemned to destroy its vital refinement-preserving property.

Yet there is a way of resolving our dilemma, but it demands a radical revision of our so far total-correctness-centred notions of specification and refinement. The key to this is in the next section.

4 General Correctness

Consider the program

$$x := 1 \; ; \textbf{ while } x \neq 0 \textbf{ do } \mathit{skip} \; [] \; x := 0 \textbf{ end} \qquad 4.0.1$$

where as before [] represents demonic choice. A total-correctness semantics tells us that when we execute the program it might not terminate but can place no constraint on the result should it do so. A partial-correctness semantics, on the other hand, tells us that if it does terminate we will have $x = 0$, but makes no judgement about whether or not that termination is inevitable. Clearly neither of these interpretations by itself accords fully with our operational intuition, which is that although termination isn't guaranteed, the result if it occurs will certainly be $x = 0$. To capture our operational intuition fully we need *both* interpretations.

As we have already noted, Dijkstra's wp yields a semantics of total correctness. It tells us under what starting conditions a program will be *guaranteed* to deliver a result satisfying a given postcondition. The less demanding semantics of partial correctness is characterised by a second predicate transformer

that Dijkstra also described in [5] but made no use of there in formulating the semantics of his guarded command language. He called this his *weakest-liberal-precondition*(wlp) predicate transformer. It tells us under what starting conditions a program can be relied on to deliver a result satisfying a given postcondition *providing it terminates at all* –that is, when it can be relied on at least not to deliver a result contradicting the given postcondition.

Totalcorrectness and partial correctness are obviously intimately linked. The difference between them is the issue of whether or not the program will terminate and so deliver any result at all. But partial correctness cannot simply be extracted from total correctness just by setting aside the question of termination. Nor can total correctness be reconstructed purely from partial correctness.

For a complete semantics of programs such as 4.0.1 which fully matches our operational intuition of their executing behaviours we need both total and partial correctness. We adopt the term *general correctness*, from Jacobs and Gries [11][2], to denote the semantics which subsumes total and partial correctness.

4.1 Linking wp and wlp

We have already written $wp(A, R)$ to represent the weakest precondition for a specification A to establish the postcondition R. Correspondingly, we write $wlp(A, R)$ to represent the weakest precondition for A to "liberally" establish R, i.e. establish R if it terminates at all. Although both these predicate transformers are positively conjunctive, which is to say they distribute through non-empty conjunctions, only $wlp(A, \_)$ is universally conjunctive, which is to say in addition it maps the empty conjunction, $true$, to itself, so that $wlp(A, true) = true$. The necessary logical relationship between wp and wlp is expressed by the pairing rule

$$wp(A, R) \quad = \quad wp(A, true) \wedge wlp(A, R) \qquad 4.1.1$$

The condition $wp(A, true)$ which appears in the rule is sometimes called the *termination predicate* of A, since it defines from where A is guaranteed to terminate with any result. We follow Abrial [1] in denoting it by $trm(A)$. It is, essentially ,A's applicability predicate. Instead of regarding $wp(A, \_)$ and $wlp(A, \_)$ as together yielding the general-correctness semantics of our specification A, we can instead regard $trm(A)$ and $wlp(A, \_)$ as fundamental. Should we need $wp(A, R)$ we can then use the pairing rule 4.1.1 to derive this as $trm(A) \wedge wlp(A, R)$.

[2] In [11] general correctness is expressed not by wp and wlp but by a different predicate transformer, gwp, defined in terms of a relational model of programs whose state space is extended by the introduction of an improper element $\perp$ to represent non-termination. However, it can easily be shown that Jacobs and Gries's ensuing semantics is equivalent to that given by wp and wlp.

4.2 Anarchic versus Fortuitous Termination

Even when termination isn't guaranteed in a computation it may still occur. Under total correctness we called this sort of termination *anarchic* since there is no constraint on the accompanying result. But in the context of general-correctness termination is never anarchic since, even when not guaranteed, its accompanying result will still be constrained by the wlp semantics embodied in the specification's before-after predicate.

For example, termination of our little example program 4.0.1, though never guaranteed, must even so if it occurs be associated with the outcome $x = 0$. Under general correctness we will therefore refer to such unguaranteed terminations as *fortuitous* rather than anarchic.

4.3 Orthogonality

We know total correctness subsumes termination within the issue of correctness of any result of the computation; the essence of general correctness is rather that it treats the termination of a computation and the correctness of its potential results as orthogonal issues. A general-correctness specification, therefore, comprises two independent elements: its termination requirement and its partial-correctness requirement.

4.4 Representing General-correctness Specifications

As we represented a total-correctness specification so we can represent a general-correctness specification by a predicate pair. The first of the pair is again simply a "unary" predicate, or *condition*, over the state space which now serves to characterise the realm of guaranteed termination of the specification. Likewise, the second is a "binary" predicate relating before-states to after-states and thus charactering which particular after-states are allowed by the specification from which particular before-states. But there is no longer any redundancy in this second predicate since all its before-states are relevant to the meaning of the specification, whether or not they fall within the ambit of the first predicate.

So in a predicate-pair representation of general-correctness specification A the first predicate embodies the termination requirement $trm(A)$ while the second embodies the partial-correctness requirement $wlp(A, \_)$. Such a predicate-pair representation of a general-correctness specification will be canonical in the sense that different predicate pairs represent different specifications.

4.5 Refinement in General Correctness

A general-correctness specification T refines another one S if it can always strictly or liberally guarantee establish any given postcondition R in circumstances where S can. We overload the $\sqsubseteq$ symbol by using it to signify general-

correctness refinement too[3]. Formally

$$S \sqsubseteq T \quad =_{df} \quad wlp(S,R) \leq wlp(T,R) \quad \text{and} \quad wp(S,R) \leq wp(T,R) \quad \text{for every postcondition } R$$

Equivalently, by virtue of the wp-wlp pairing rule 4.1.1

$$S \sqsubseteq T \quad = \quad trm(S) \leq trm(T) \quad \text{and} \quad wlp(S,R) \leq wlp(T,R) \quad \text{for every postcondition } R$$

4.6 Prescriptions

In [7] we formulated a general-correctness analogy of Hoare and He's designs which we called a *prescription* Like a design, a prescription is expressible as a predicate pair, which we denote as $P \vdash\!\!\vdash Q$. This can be informally interpreted as

> If the program starts in circumstances satisfying P, it must terminate. Moreover should termination ensue, the program having started whether in circumstances satisfying P or not, then Q will be satisfied.

The crucial distinction between this and our earlier interpretation of the design $P \vdash Q$ is therefore that, while termination is still only guaranteed if P holds when we start the program, we now have an unqualified assurance that Q will be satisfied upon any termination, whether inevitable or merely fortuitous. We can easily extract the trm and wlp from a prescription $P \vdash\!\!\vdash Q$ on a state space w by

$$trm(P \vdash\!\!\vdash Q) \quad = \quad P \qquad 4.6.1$$

$$wlp(P \vdash\!\!\vdash Q\ ,\ R) \quad = \quad \forall\, w' \bullet Q \Rightarrow R' \qquad 4.6.2$$

A prescription, and indeed by extension any predicate-pair general-correctness specification, enjoys the significant advantage of being a canonical form. That is to say, given prescriptions $P_1 \vdash\!\!\vdash Q_1$ and $P_2 \vdash\!\!\vdash Q_2$ over the same alphabet, it is easy to show that

$$P_1 \vdash\!\!\vdash Q_1 \quad = \quad P_2 \vdash\!\!\vdash Q_2 \quad \text{if and only if} \quad P_1 \;=\; P_2 \;\text{ and }\; Q_1 \;=\; Q_2$$

This means we can safely define general-correctness specification composition operators in terms of predicate-pair representations of those specifications without encountering the same problem of ill-definedness we saw for total-correctness specifications in Section 2.4. Also, it follows from properties 4.6.1 and 4.6.2 that a general-correctness predicate-pair specification such as the prescription $P_1 \vdash\!\!\vdash Q_1$ is refined by another one $P_2 \vdash\!\!\vdash Q_2$ precisely if

$$P_1 \leq P_2 \quad \text{and} \quad Q_2 \leq Q_1$$

[3] The resulting potential ambiguity in the assertion $S \sqsubseteq T$ is resolved by the context which determines whether S and T are total- or general-correctness specifications.

4.7 Guaranteed Non-termination

An important advantage which general correctness holds over total correctness is that it provides a semantic framework powerful enough in which to express a guarantee of non-termination as well as of termination. As an extreme example, the prescription $false \vdash\!\!\vdash false$ precisely describes the infinite loop program which never terminates. (In contrast the design $false \vdash false$ is synonymous with $false \vdash true$. It describes what is commonly known as *abort*, and can thus terminate anarchically.)

More generally, a prescription of the form $P \vdash\!\!\vdash P \wedge Q$ if executed in circumstances when P holds will definitely terminate with a result defined by Q. If executed when P doesn't hold it must fail to terminate. Such an ability to express non-termination requirements is essential for specifying members of the important class of computations comprised by partial decision procedures[4] [4].

5 Junctive Compositions in General Correctness

We now formulate the general-correctness analogues of our previous four junctive compositions of total-correctness specifications. We will express these in the notation of our prescriptions, assumed to share a common alphabet of variables. Once more we restrict our attention to binary compositions, though again each could be generalised in an obvious way. The four resulting compositions of the prescriptions $P_1 \vdash\!\!\vdash Q_1$ and $P_2 \vdash\!\!\vdash Q_2$ are shown in Table 2. We overload our four composition operator symbols now to represent these general-correctness compositions as well as their earlier total-correctness counterparts. The overloadingis resolved by the nature of the operands as either total- or general-correctness specifications.

Besides being well-defined, the four compositions of Table 2 are also all monotonic with respect to general-correctness refinement. We can immediately see that all four compositions of Table 2 have a uniformly simple form, unlike the last two ones of Table 1 whose definitions were complicated by the need to ensure well-definedness. The first three operators of Table 2 have similar operational interpretations to those of Table 1. The demonic choice [] once again represents a choice of executions vested in a demon inhabiting the machine on which our program is being executed.

The fusion operator $\odot$ now constrains the nondeterminism of outcomes from both guaranteed and fortuitous terminations of each of its operands, so as to achieve an agreed outcome, and where this is impossible because intersection of those outcomes is empty it behaves miraculously if termination is jointly guaranteed, and if not it doesn't terminate. Where either operand isn't guaranteed to terminate neither is their fusion. For example, we calcu-

[4] A partial decision procedure is guaranteed either to terminate with a correct result or fail to terminate at all, but it won't terminate anarchically with an incorrect result.

symbol	name	definition
$[]$	demonic choice	$P_1 \wedge P_2 \vdash\!\!\vdash Q_1 \vee Q_2$
$\odot$	fusion	$P_1 \wedge P_2 \vdash\!\!\vdash Q_1 \wedge Q_2$
$\diamond$	join	$P_1 \vee P_2 \vdash\!\!\vdash Q_1 \wedge Q_2$
$\#$	concert	$P_1 \vee P_2 \vdash\!\!\vdash Q_1 \vee Q_2$

Table 2
General-correctness junctive compositions of $P_1 \vdash\!\!\vdash Q_1$ with $P_2 \vdash\!\!\vdash Q_2$

late the fusion of $true \vdash\!\!\vdash x' = 7$ respectively with $true \vdash\!\!\vdash x' = 11$ and then $false \vdash\!\!\vdash x' = 11$:

$$\begin{array}{lcl} true \vdash\!\!\vdash x' = 7 \ \odot \ true \vdash\!\!\vdash x' = 11 & = & true \wedge true \vdash\!\!\vdash x' = 7 \wedge x' = 11 \\ & = & true \vdash\!\!\vdash false \\ & = & magic \end{array}$$

while

$$\begin{array}{lcl} true \vdash\!\!\vdash x' = 7 \ \odot \ false \vdash\!\!\vdash x' = 11 & = & true \wedge false \vdash\!\!\vdash x' = 7 \wedge x' = 11 \\ & = & false \vdash\!\!\vdash false \end{array}$$

which specifies an infinite loop.

The join operator $\diamond$ is once again the least co-refinement of the two specifications concerned, though of course this time the refinement concerned is general-correctness refinement.

The concert operator # is more interesting here. Under general correctness it seems to come into its own, and has in fact already been described in such a context by Dunne *et al* [8,9], albeit not in terms of prescriptions. Once again our operational intuition of # is that it represents parallel execution of its two operands on disjoint copies of the global state in a termination pact: the two executions proceed until either terminates, upon which the resulting actual global state is recovered entirely from the terminating one's copy. The other execution is aborted, its copy of the global state being discarded. But general correctness abolishes the threat of anarchic termination, and in its absence our general-correctness # is a more potent and useful composition than its total-correctness counterpart in Table 1.

6 Applications of General-correctness Concert

As an example of the use of our general-correctness concert composition #, suppose we wish to implement a prescription C, where

$$C \quad = \quad true \ \vdash\!\!\!\vdash \ Q$$

Let P_1 and P_2 be complementary conditions on our state space, such that

$$P_1 \vee P_2 \ = \ true$$

and consider the following prescriptions A and B, where

$$A \quad = \quad P_1 \ \vdash\!\!\!\vdash \ Q$$

$$B \quad = \quad P_2 \ \vdash\!\!\!\vdash \ Q$$

Then we have that

$$\begin{array}{rcl} A \# B & = & P_1 \vdash\!\!\!\vdash Q \ \ \# \ \ P_2 \vdash\!\!\!\vdash Q \\ & = & P_1 \vee P_2 \ \vdash\!\!\!\vdash \ Q \vee Q \\ & = & true \ \vdash\!\!\!\vdash \ Q \\ & = & C \end{array}$$

Thus we can implement C, should we choose, as the concert of A and B.

6.1 Concerted Proof-search

As we pointed out in [9], our concert models the sort of parallel search strategy we might employ in an automated prover. For example, our prover could attempt to resolve a goal G one way or the other by embarking simultaneously on proofs of both G and $\neg\ G$. We assume, of course, that both subordinate proof-searchers can be relied on to terminate only after finding their proofs and otherwise to carry on searching indefinitely.

6.2 Evaluation of Logical expressions with Undefined Terms

We can employ a similar strategy to evaluate a conjunction or disjunction of logical operands where, notwithstanding that because of undefined terms evaluation of some operands may not terminate, the overall logical expression can still be evaluated. Such a philosophy inspires VDM's *Logic of Partial Functions* (LPF). Jones [12] gives truth tables in this logic for the familiar logical operators of conjunction, disjunction and negation. In addition to the ordinary truth-values true and false, these feature the symbol $*$ to indicate undefinedness. Jones says

> It is useful to think of [each of] these operators being evaluated by a program which has access to the parallel evaluation of its operands. As soon as

a result is available for one operand, it is considered; if the single result determines the overall result (e.g. one true for a disjunction), evaluation ceases and the (determined) result is returned.

Let $true(B)$ be a partial evaluation procedure for the boolean expression B which terminates if B is true and otherwise (i.e. if B is false or undefined) doesn't terminate, and which in any case has no side-effects on the global state v. Likewise, let $false(B)$ be a partial evaluation procedure without side-effects which terminates if B is false and otherwise doesn't terminate. More formally, if δ is a metapredicate on expressions such that δE holds if and only if the expression E is well-defined, then we can specify these procedures by

$$\begin{array}{lll} true(B) & =_{df} & \delta B \wedge B \ \vdash\!\vdash \delta B \wedge B \wedge v' = v \\ false(B) & =_{df} & \delta B \wedge \neg B \ \vdash\!\vdash \delta B \wedge \neg B \wedge v' = v \end{array}$$

It then follows directly from our definitions of $\odot$ and $\#$ that

$$\begin{array}{lll} true(B) \odot false(B) & = & false \ \vdash\!\vdash false \\ true(B) \# false(B) & = & \delta B \ \vdash\!\vdash \delta B \wedge v' = v \end{array}$$

Thus for any boolean expression B, the fusion $true(B) \odot false(B)$ never terminates, while the concert $true(B) \# false(B)$ terminates precisely when B is well-defined.

We can now characterise the three fundamental logical operators of Jones's LPF in terms of of our two partial evaluation procedures:

$$\begin{array}{lcl} true(A \vee_{\mathtt{LPF}} B) & = & true(A) \# true(B) \\ false(A \vee_{\mathtt{LPF}} B) & = & false(A) \odot false(B) \\ \\ true(A \wedge_{\mathtt{LPF}} B) & = & true(A) \odot true(B) \\ false(A \wedge_{\mathtt{LPF}} B) & = & false(A) \# false(B) \\ \\ true(\neg_{\mathtt{LPF}} A) & = & false(A) \\ false(\neg_{\mathtt{LPF}} A) & = & true(A) \end{array}$$

From these we can derive similar characterisations of derived operators like implication and bi-implication. For example, since $A \Rightarrow B$ is an abbreviation for $\neg A \vee B$, we can show

$$\begin{array}{lcl} true(A \Rightarrow_{\mathtt{LPF}} B) & = & false(A) \# true(B) \\ false(A \Rightarrow_{\mathtt{LPF}} B) & = & true(A) \odot false(B) \end{array}$$

Similarly, treating $A \Leftrightarrow B$ as an abbreviation for $(A \Rightarrow B) \wedge (B \Rightarrow A)$, and given that $\#$ and $\odot$ distribute through each other, we can show

$$\begin{array}{lcl} true(A \Leftrightarrow_{\mathtt{LPF}} B) & = & (false(A) \odot false(B)) \# (true(A) \odot true(B)) \\ false(A \Leftrightarrow_{\mathtt{LPF}} B) & = & (true(A) \odot false(B)) \# (false(A) \odot true(B)) \end{array}$$

7 Conclusion

We have explored our family of junctive compositions of specifications, first in the Refinement Calculus's conventional context of total correctness and then in the rather more esoteric one of general correctness. We have seen that this second context arguably provides a better basis for formulating such compositions, since in it they can be expressed more uniformly and simply than their counterparts in total correctness. The interpretations of our general-correctness compositions are compatible with their total-correctness counterparts, and indeed in some cases, as most notably with our concert, more tractable and so more useful. In particular, we have shown how our general-correctness concert can be used to implement a decidable specification by a pair of semi-decidable ones.

Finally, we note that in this paper we have restricted our attention to compositions of specifications sharing a common reference frame of variables, and without any restriction on their update frames. In reality we frequently want to compose specifications with different update frames or even different reference frames. A practicable specification structuring environment therefore needs composition mechanisms which take frames fully into account.

Acknowledgement

I am indebted to the anonymous reviewer whose extensive comments on a previous effort of mine directed at a different publication venue were partly instrumental in inspiring me to write this paper. In particular, that reviewer drew my attention to several other authors' works now referenced here.

References

[1] Abrial, J.-R., "The B-Book: Assigning Programs to Meanings," Cambridge University Press, 1996.

[2] Back, R.-J., *A calculus of refinements for program derivations*, Acta Informatica **25** (1988), pp. 593–624.

[3] Back, R.-J. and M. Butler, *Fusion and simultaneous execution in the refinement calculus*, Acta Informatica **35** (1998), pp. 921–940.

[4] Cutland, N., "Computability: an introduction to recursive function theory," Cambridge University Press, 1980.

[5] Dijkstra, E., "A Discipline of Programming," Prentice-Hall International, 1976.

[6] Dunne, S., *A theory of generalised substitutions*, in: D. Bert, J. Bowen, M. Henson and K. Robinson, editors, *ZB2002: Formal Specification and Development in Z and B*, number 2272 in Lecture Notes in Computer Science (2002), pp. 270–290.

[7] Dunne, S., *Recasting the Hoare-He unifying theory of programs in the context of general correctness*, in: A. Butterfield and C. Pahl, editors, *Proceedings of the 5th Irish Workshop in Formal Methods (IWFM'01)*, July 2001, Trinity College Dublin.

[8] Dunne, S., W. Stoddart and A. Galloway, *Hypersubstitutions: extending the generalised substitution to model semi-decidable operations*, in: H. Habrias, editor, *The First B Conference* (1996), pp. 221–235, isbn 2-906082-25-22.

[9] Dunne, S., W. Stoddart and A. Galloway, *Specification and refinement in general correctness*, in: A. Evans, D. Duke and A. Clark, editors, *Proceedings of the 3rd Northern Formal Methods Workshop* (1998), http://www.ewic.org.uk/ewic/workshop/view.cfm/NFM-98.

[10] Hoare, C. and H. Jifeng, "Unifying Theories of Programming," Prentice Hall, 1998.

[11] Jacobs, D. and D. Gries, *General correctness: a unification of partial and total correctness*, Acta Informatica **22** (1985), pp. 67–83.

[12] Jones, C., "Systematic Software Development Using VDM (2nd edn)," Prentice-Hall, 1990.

[13] Leino, K. and R. Manohar, *Joining specifications*, Theoretical Computer Science **216** (1999), pp. 375–394.

[14] Mahony, B., *The least conjunctive refinement and promotion in the refinement calculus*, Formal Aspects of Computing **11** (1999), pp. 75–105.

[15] Morgan, C., *The specification statement*, ACM Transactions on Programming Languages and Systems **10** (1988).

[16] Morgan, C., "Programming from Specifications (2nd edn)," Prentice Hall International, 1994.

[17] Morris, J., *A theoretical basis for stepwise refinement and the programming calculus*, Science of Computer programming **9** (1987), pp. 287–306.

[18] Spivey, J., "The Z Notation: a Reference Manual (2nd edn)," Prentice Hall, 1992.

[19] Ward, N., *Adding specification constructs to the refinement calculus*, in: J. Woodcock, editor, *FME'93: Industrial-strength Formal Methods*, number 670 in Lecture Notes in Computer Science (1993), pp. 652–670.

[20] Woodcock, J. and J. Davies, "Using Z: Specification, Refinement and Proof," Prentice Hall, 1996.